ERROR CORRECTION DECODER USING CELLS WITH PARTIAL SYNDROME GENERATION

BACKGROUND

1. Field of the Invention

[0001] The present application generally relates to error correction of data using error correction codes (e.g., Bose-Chaudhuri-Hocquenghem (BCH), Reed Solomon codes, and the like), and more particularly to an error correction decoder using cells with partial syndrome generation.

2. Related Art

[0002] Error correction of data may be used in various applications, such as data storage devices, telecommunication systems, and the like. For example, in a data storage device, data is stored by writing the data to a storage medium in the storage device. The stored data can be later retrieved from the storage device by reading the data from the storage medium. However, for a number of reasons, an error may exist in the data retrieved from the storage device, meaning that the stored data cannot be retrieved or is different from the data originally stored on the storage medium. For example, a portion of the stored data on the storage medium may degenerate over time such that the portion of the stored data cannot be properly read at a later time.

[0003] Conventional error correction techniques include generating or encoding one or more redundancy blocks for data, which can be used in a decoding process to correct errors in the data. Typically the decoding process is performed using specialized hardware, which tends to be complicated and difficult to modify.

SUMMARY

[0004] In one exemplary embodiment, a decoder to correct errors in data includes a plurality of cells. Each cell generates a partial syndrome based on data blocks and one or more redundancy blocks. Each cell generates a partial error value based on a portion of an inverse of an error location matrix that identifies locations of any data blocks having errors. A summation logic connected to the plurality of cells generates an error

value based on the partial error values generated by the plurality of cells. The error value corrects errors in a data block having errors.

BRIEF DESCRIPTION OF THE FIGURES

[0005] Fig. 1 depicts an exemplary host terminal connected to an exemplary storage device;

[0006] Fig. 2 depicts an exemplary entity having a set of data blocks, redundancy blocks, and cyclic redundancy checking codes;

[0007] Fig. 3 depicts portions of the exemplary entity of Fig. 2;

[0008] Fig. 4 depicts an exemplary decoder;

[0009] Fig. 5 depicts an exemplary cell of the exemplary decoder depicted in Fig. 4;

[0010] Fig. 6 depicts an exemplary read process performed by the exemplary cell depicted in Fig. 5;

[0011] Fig. 7 depicts a portion of the exemplary cell depicted in Fig. 5;

[0012] Fig. 8 depicts an exemplary write process performed by the exemplary cell depicted in Fig. 5, and

[0013] Fig. 9 depicts another portion of the exemplary cell depicted in Fig. 5.

DETAILED DESCRIPTION

[0014] The following description sets forth numerous specific configurations, parameters, and the like. It should be recognized, however, that such description is not intended as a limitation on the scope of the present invention, but is instead provided to provide a better description of exemplary embodiments.

[0015] For the sake of example, error correction of data in a storage device is described below. It should be recognized, however, that error correction can be used in various applications, including telecommunications.

[0016] With reference to Fig. 1, a host terminal 102 is depicted connected to a storage device 104. Host computer 102 can be any type of computer, such as a personal computer, a workstation, a server, and the like. Storage device 104 can be any type of storage drive, such as a tape drive, a hard drive, and the like. It should be recognized that host terminal 102 can be connected to any number of storage devices 104, and any number of host terminals 102 can be connected to one or more storage devices 104.

[0017] With continued reference to Fig. 1, in one exemplary embodiment, storage device 104 is configured to detect and correct errors in data stored in storage device 104. More specifically, when data stored in storage device 104 is retrieved, storage device 104 is configured to use redundancy blocks, which are also referred to as error correction code (ECC) redundancy blocks, to correct errors in the retrieved data, such as if the retrieved data is different from the data that was originally stored in storage device 104 or if the stored data cannot be retrieved. Additionally, inner codes, such as cyclic redundancy checking (CRC) codes, can be used to detect errors in the retrieved data. However, it should be recognized that error correction codes, such as Reed-Solomon codes, can be used to detect as well as correct errors.

[0018] In the embodiment depicted in Fig. 1, storage device 104 includes a storage medium 106, a channel and read/write head 108, a processor 110, and an error detection/correction unit 112. In storage device 104, data is stored in storage medium 106. Read/write head 108 reads and/or writes data to storage medium 106. Processor 110 controls the operation of storage device 104, including the operation of channel and read/write head 108. As will be described in greater detail below, error detection/correction unit 112 detects and corrects errors in data stored in storage medium 106.

[0019] In the present exemplary embodiment, error detection/correction unit 112 includes a data buffer 114, a redundancy block encoder/decoder 116, and an inner code

encoder/decoder 118. When data is to be stored on storage medium 106, data is received from host terminal 102 and written to data buffer 114. Redundancy block encoder/decoder 116 generates redundancy blocks for data in data buffer 114. Inner code encoder/decoder 118 generates inner codes (e.g., CRC codes, Reed-Solomon codes, and the like) for data in data buffer 114. Read/write head 108 then writes the data and the generated redundancy blocks and inner codes to storage medium 106.

[0020] When data is to be read from storage medium 106, read/write head 108 reads data, redundancy blocks, and inner codes from storage medium 106 to data buffer 114. As will be described in greater detail below, any errors in the data read from storage medium 106 are detected and corrected using the inner codes and redundancy blocks. The data may then be transferred to host terminal 102.

[0021] In the present exemplary embodiment, data is transferred between host terminal 102 and storage device 104 in data records, which are stored in a buffer. The data records are divided into data blocks of a predetermined length, such as two kBytes, four kBytes, six kBytes, and the like. It should be recognized, however, that data blocks of various lengths may be used.

[0022] After data blocks are retrieved from storage medium 106, retrieved data blocks having errors are detected, where an error in a retrieved data block indicates that the data in the retrieved data block cannot be read or is different than the data in the data block when the data block was originally stored in storage medium 106. For example, CRC codes can be used to detect when the data in the retrieved data block is different from the data in the data block when the data was originally stored in storage medium 106. More specifically, prior to storing a data block in storage medium 106, a CRC code is generated for the data block and stored with the data block in storage medium 106. When the data block is later retrieved, a new CRC code is generated for the retrieved data block. The new CRC code is then compared to the CRC code retrieved from storage medium 106, which corresponds to the retrieved data block and was originally generated for the retrieved data block before storing the retrieved data block in storage medium 106. If the new CRC code and the retrieved CRC code differ, then an error is detected

for that data block. It should be recognized, however, that various types of error detection codes, including Reed-Solomon codes, may be used.

[0023] In the present exemplary embodiment, an error location matrix identifies the location of any data retrieved from the storage medium having errors. For example, an exemplary error location matrix can be expressed as:

$$\begin{bmatrix} 1 & X_0 & \cdots & X_0^{\rho-1} \\ 1 & X_1 & & X_1^{\rho-1} \\ \vdots & & \ddots & \vdots \\ 1 & X_{\rho-1} & \cdots & X_{\rho-1}^{\rho-1} \end{bmatrix},$$

where X is the location of an error and ρ is the total number of data blocks and redundancy blocks in error.

[0024] In the present exemplary embodiment, the inverse of the error location matrix is generated. For example, the inverse of the exemplary error location matrix in the above example can be expressed as follows:

$$\begin{bmatrix} U_{0,0} & U_{1,0} & \cdots & U_{\rho-1,0} \\ U_{0,1} & U_{1,1} & & U_{\rho-1,1} \\ \vdots & & \ddots & \vdots \\ U_{0,\rho-1} & U_{1,\rho-1} & \cdots & U_{\rho-1,\rho-1} \end{bmatrix}.$$

Note that the error location matrix is a Vandermonde matrix, which in general requires $O(\rho^2)$ operations to compute the inverse, in comparison to general matrix inversions, which requires $O(\rho^3)$ operations. A row in a Vandermonde matrix consists of a constant value raised to incrementing powers, starting at zero. The inverse of the error location matrix can by generated by firmware or hardware using various matrix inversion techniques.

[0025] In the present exemplary embodiment, redundancy blocks are used to correct errors in the retrieved data blocks. More specifically, prior to storing data blocks in storage medium 106, redundancy blocks are generated based on the data blocks, and stored with the data blocks in storage medium 106. When the data blocks are later

retrieved, data blocks identified as having errors are corrected using the redundancy blocks.

[0026] In the present exemplary embodiment, redundancy blocks are Bose-Chaudhuri-Hocquenghem (BCH) codes, and more particularly Reed-Soloman codes. For a more detailed description of Reed-Solomon codes, see Peterson & Weldon, Error Correcting Codes, 2d Edition, MIT Press, 1972, which is incorporated in its entirety herein by reference. It should be recognized, however, that various types of error correction codes may be used.

[0027] In the present exemplary embodiment, a set of data blocks, a set of redundancy blocks, and a set of redundancy symbols of an inner code are read and written together as a group referred to as an "entity." For example, with reference to Fig. 2, an entity 202 is depicted having 16 data blocks 204, four redundancy blocks 206, and 20 redundancy symbols of an inner code 208. It should be recognized, however, that entity 202 can include various numbers of data blocks 204, redundancy blocks 206, and redundancy symbols of an inner code 208. For example, entity 202 can include 32 data blocks 204 and eight redundancy blocks 206, 112 data blocks 204 and 16 redundancy blocks 206, and the like. Additionally, redundancy symbols of an inner code 208 can both detect and correct error within a data block 204 or redundancy block 206.

[0028] Fig. 2 depicts the form in which entity 202 can be stored in data buffer 114 (Fig. 1). It should be recognized, however that entity 202 need not exist physically in the form depicted in Fig. 2. It should also be recognized that data in entity 202, and more particularly the data in a data block 204, need not correspond to a single file. Instead, in the present exemplary embodiment, data received from host terminal 102 (Fig. 1) is interleaved. As such, the data in a particular data block 204 can correspond to portions of separate files received from host terminal 102 (Fig. 1).

[0029] Fig. 2 also depicts logical relationships between data blocks 204, redundancy blocks 206, and redundancy symbols of an inner code 208 of entity 202. With reference to Fig. 3, portions of entity 202 are shown in greater detail to more clearly

illustrate the logical relationships between data blocks 204, redundancy blocks 206, and redundancy symbols of an inner code 208.

[0030] In Fig. 3, redundancy symbols of an inner code 208 are depicted as being CRC codes. It should be recognized, however, that various types of error detection or error correction codes can be used, such as Reed-Solomon codes.

[0031] In the present exemplary embodiment, redundancy symbols of an inner code 208 corresponds to a data block 204 or redundancy block 206 and is used to detect an error in data block 204 or redundancy block 206. For example, CRC code CRC_{19} corresponds to data block D_{19} of entity 202. Thus, to detect an error in data block D_{19}, after retrieving data block D_{19} from storage medium 106 (Fig. 1), a new CRC code CRC_{19}' is generated for retrieved data block D_{19}. The new CRC code CRC_{19}' is then compared to the CRC code retrieved from storage medium 106 (Fig. 1) corresponding to retrieved data block D_{19} (i.e., CRC code CRC_{19}). If the new CRC code CRC_{19}' and the retrieved CRC code CRC_{19} differ, then an error is detected for data block D_{19}.

[0032] In the present exemplary embodiment, the number of redundancy blocks determines the maximum number of data blocks that can be corrected. Thus, in the example depicted in Fig. 2, a total of four redundancy blocks 206 may be used to correct a maximum of four data blocks 204 in error.

[0033] In the present exemplary embodiment, each redundancy block 204 is generated based on the data in all of the data blocks of entity 202. For example, redundancy blocks E_0, E_1, E_2, and E_3 are each generated based on the data in data blocks D_4, D_2, ..., and D_{19}. As described above, with reference to Fig. 1, redundancy blocks can be generated by redundancy block encoder/decoder 116. As also described above, redundancy blocks are initially generated for data received from host terminal 102. The generated redundancy blocks and the received data are then stored in storage medium 106.

[0034] With reference again to Fig. 3, although redundancy blocks E_0, E_1, E_2, and E_3 are generated based on the same set of data (i.e., data blocks 204 of entity 202), each

redundancy block 206 is unique as to each other. More specifically, in the present embodiment, redundancy blocks E_0, E_1, E_2, and E_3 are Bose-Chaudhuri-Hocquenghem (BCH) codes, and more particularly Reed-Solomon codes.

[0035] With reference to Fig. 1, in the present exemplary embodiment, redundancy block encoder/decoder 116 operates as an encoder to generate redundancy blocks to be stored with data blocks in storage medium 106. Redundancy block encoder/decoder 116 operates as a decoder to correct errors in data blocks retrieved from storage medium 106. It should be recognized, however, that redundancy block encoder/decoder 116 can be implemented as separate components (i.e., an encoder component and a decoder component) in storage device 104.

[0036] With reference now to Fig. 4, an exemplary encoder 400 to correct errors in data blocks retrieved from a storage medium is depicted. As noted above, encoder 400 can be implemented as an integrated part of redundancy block encoder/decoder 116 (Fig. 1) or as a separate component within storage device 104 (Fig. 1).

[0037] As depicted in Fig. 4, encoder 400 includes a plurality of cells 402 and a summation logic 404 connected to the plurality of cells 402. Each cell 402 generates a partial syndrome based on the data blocks and redundancy blocks retrieved from the storage medium. Each cell 402 also generates a partial error value based on a portion of the inverse of an error location matrix that identifies the locations of any data blocks retrieved from the storage medium having errors. As described above, the inverse of the error location matrix can be generated by firmware or hardware. Summation logic 404 generates an error value, which corrects errors in a data block retrieved from the storage medium, based on the partial error values generated by plurality of cells 402.

[0038] With reference to Fig. 1, as described above, data blocks and redundancy blocks retrieved from storage medium 106 are held in data buffer 114. More particularly, with reference to Fig. 2, in one exemplary embodiment, data blocks 204 and redundancy blocks 206 can be stored in data buffer 114 (Fig. 1) in the form of entity 202.

[0039] With reference again to Fig. 4, in the present exemplary embodiment, plurality of cells 402 are connected to data buffer 114 (Fig. 1) through lines 406. Each cell 402 receives as an input a portion of entity 202 (Fig. 2) stored in data buffer 114 (Fig. 1). More particularly, each cell 402 reads in portions of entity 202 (Fig. 2) in cache bursts.

[0040] With reference to Fig. 2, in the present exemplary embodiment, each cache burst is a portion of a single data block 204 or redundancy block 206. Additionally, in the present exemplary embodiment, portions of data blocks 204 and redundancy blocks 206 are read in a raster pattern.

[0041] For example, assume that each cache burst is 32 bytes long. The first 32 bytes of the first data block 204 is read in a first cache burst. The first 32 bytes of each of the next 15 data blocks 204 and four redundancy blocks 206 of entity 202 are then read in 19 subsequent cache bursts. After the first 32 bytes of the last redundancy block 206 is read, the second subsequent 32 bytes of the first data block 204 is read, and then the second subsequent 32 bytes of each of the next 15 data blocks 204 and four redundancy blocks 206 of entity 202 are read. In this manner, data blocks 204 and redundancy blocks 206 of entity 202 are read in cache bursts in a raster pattern in 32 byte portions. It should be recognized, however, that the size of the cache burst can vary, and data blocks 204 and redundancy blocks 206 can be read in various patterns. For example, each cache burst can be 64 bytes rather than 32 bytes.

[0042] In the present exemplary embodiment, a portion of each data block 204 and redundancy block 206 in entity 202 can be logically grouped as a code word 210. For example, the first portions of each data block 204 and redundancy block 206 can be logically grouped as a first code word 210, which corresponds to the first column in Fig. 2. The second byte of each data block 204 and redundancy block 206 can be logically grouped as a second code word 210, which corresponds to the second column in Fig. 2.

[0043] In the present exemplary embodiment, code word 210 is one byte wide. Thus, if data blocks 204 and redundancy blocks 206 are 2 kBytes long, then data blocks 204 and redundancy blocks 206 in entity 202 can be logically grouped into 2,000

separate, individual code words 210. Additionally, if data blocks 204 and redundancy blocks 206 are read using a cache burst of 32 bytes, which corresponds to 16 long words of 32 bits each, then portions of 32 code words 210 are read at a time.

[0044] With reference again to Fig. 4, in the present exemplary embodiment, the number of cells 402 in decoder 400 corresponds to the number of redundancy blocks 206 (Fig. 2) in entity 202, which in turn corresponds to the maximum number of data blocks 204 that can be corrected. Thus, decoder 400 can be modified based on the maximum number of data blocks 204 to be corrected. For example, when a maximum of four data blocks 204 are to be corrected, decoder 400 is modified to include four cells 402. Similarly, when a maximum of 16 or 32 data blocks 204 are to be corrected, decoder 400 is modified to include 16 or 32 cells 402.

[0045] With reference now to Fig. 5, an exemplary cell 402 is depicted. In the present exemplary embodiment, each cell 402 includes an input logic 502, a queue 504, a multiplier 506, a multiplexer 508, and queues 510 and 512.

[0046] As described above, each cell 402 generates a partial syndrome based on data blocks and one or more redundancy blocks retrieved from the storage medium. With reference to Fig. 6, an exemplary read process 600 of generating a partial syndrome in cell 400 (Fig. 4) is depicted.

[0047] In 602, cell 402 (Fig. 4) is set to a read mode. With reference to Fig. 5, in the present exemplary embodiment, input logic 502 is set to receive inputs from line 406, which is connected to data buffer 114 (Fig. 1), and feedback line 514, which is connected to the output of multiplier 506. More particularly, with reference to Fig. 7, multiplexer 702 is set to receive input from an XOR gate 704 connected to line 406 and feedback line 514. With reference to Fig. 5, multiplier 506 is set to receive inputs from queue 504, which holds intermediate results generated by input logic 502, and queue 510, which holds a root of a generator polynomial. More particularly, multiplexer 508 with inputs connected to queues 510 and 512 is set to receive input from queue 510.

[0048] With reference to Fig. 6, in 604, cell 402 (Fig. 4) reads data from data buffer 114 (Fig. 1). In the present exemplary embodiment, cell 402 (Fig. 4) reads in a portion of a data block or a redundancy block from entity 202 (Fig. 2) stored in data buffer 114 (Fig. 1) as a cache burst.

[0049] For example, data can be read from data buffer 114 (Fig. 1) in a cache burst of 32 bytes, which corresponds to 16 long words. With reference to Fig. 5, line 406 and feedback line 514 are 32 bits wide, which allows cell 402 to process 32 bits (one long word) at a time. It should be recognized, however, that lines 406 and feedback line 514 can be any size, and cell 402 can process any number of bits at a time.

[0050] In 606, the data read from data buffer 114 (Fig. 1) is summed with the output from multiplier 506 (Fig. 5). With reference to Fig. 5, in the present exemplary embodiment, input logic 502 performs an exclusive-OR (XOR) operation on a cache burst read from entity 202 (Fig. 2) through line 406 and the output of multiplier 506 through feedback line 514. As described above, with reference to Fig. 7, input logic 502 includes XOR gate 704, which can perform the XOR operation. It should be recognized, however, that input logic 502 can include various components, including various types and numbers of logic gates, to sum the data read from data buffer 114 (Fig. 1) with the output from multipliers 504 (Fig. 5).

[0051] With reference to Fig. 6, in 608, the sum of the data read from data buffer 114 (Fig. 1) and the output from multiplier 506 (Fig. 5) is stored as an intermediate result. With reference to Fig. 5, in the present exemplary embodiment, the intermediate result generated by input logic 502 is stored in queue 504.

[0052] In the present exemplary embodiment, the size and number of entries in queue 504 is determined based on the size of the cache burst used to read data from data buffer 114 (Fig. 1) and the size of line 406 and feedback line 514. For example, if data is read from data buffer 114 (Fig. 1) in a cache burst of 32 bytes, which corresponds to 16 long words, and line 516 and feedback line 514 are 32 bits wide, then each long word in the cache burst is summed with the output from multiplier 506 and stored as an entry in queue 504. Thus, in this example, queue 504 includes eight entries with each entry 32

bits long. If the size of the cache burst is changed, then the size of queue 504 can also be changed. For example, if the cache burst is 64 bytes long, then queue 504 can include 16 entries with each entry 32 bits long.

[0053] With reference to Fig. 6, in 610, the intermediate result is multiplied with a root of a generator polynomial. With reference to Fig. 5, in the present exemplary embodiment, multiplier 506 is a Galois Field Multiplier that performs a Galois Field multiplication between the intermediate result stored in queue 504 and a root of a generator polynomial stored in queue 510.

[0054] In the present exemplary embodiment, multiplier 506 performs multiple, parallel Galois Field multiplications. For example, if each entry of queue 504 is 32 bits long, the 32 bits of data from an entry of queue 504 can be formatted as four parallel eight bit Galois Field elements. Multiplier 506 can then perform four parallel eight-by-eight Galois Field multiplications between the 32 bit long entry from queue 504 and an eight bit root of a generator polynomial from queue 510.

[0055] With reference to Fig. 6, in 612, if all of the data blocks and redundancy blocks of entity 202 (Fig. 2) have not been processed, loop 604 to 612 is iterated to process the next data block or redundancy block. In the present exemplary embodiment, if the next data block to be processed was determined to have errors, then in 604 all zeros are read rather than reading a cache burst from data buffer 114 (Fig. 1). If all of the data blocks and redundancy blocks have been processed, read process 600 terminates at 614.

[0056] In the present exemplary embodiment, the number of iterations of loop 604 to 612 corresponds to the number of data blocks and redundancy blocks in entity 202 (Fig. 2). For example, with reference to Fig. 3, assume that portions of entity 202 are read in cache bursts of 32 bytes in length. In a first iteration of loop 604 to 612 (Fig. 6), a first cache burst containing the first 32 bytes of data block D_{19} is read and processed. In a second iteration of loop 604 to 612 (Fig. 6), a second cache burst containing the first 32 bytes of data block D_{18} is read and processed. Assume that data block D_4 was identified as having errors. Thus, in a 16^{th} iteration of loop 604 to 612 (Fig. 6), all zeros are read and processed rather than reading in a cache burst from data block D_4. In this example,

loop 604 to 612 (Fig. 6) is iterated 20 times to process all 16 data blocks and four redundancy blocks.

[0057] With reference to Fig. 5, note that in the first iteration of loop 604 to 612 (Fig. 6), the result of the sum of the first cache burst and the output of the multiplier 506 is the data read in the first cache burst because queue 504 is empty and the output of multiplier 506 is zero. In the second iteration, queue 504 holds the data read in the first cache burst, and the output of multiplier 506 is the data read in the first cache burst multiplied by a root of a generator polynomial. Thus, the result of the sum of the second cache burst and the output of the multiplier 506 is the sum of the data read in the second cache burst and the result of multiplying the data read in the first cache burst by a root of a generator polynomial. This result is then stored in queue 504 as the new intermediate result. When read process 600 is terminated in 614 (Fig. 6) after all of the data blocks and redundancy blocks have been processed, the final result stored in queue 504 of each cell 402 is the partial syndrome for each cell 402.

[0058] With reference again to Fig. 4, in the present exemplary embodiment, each cell 402 uses a different root of a generator polynomial. Thus, different partial syndromes are generated in each of the plurality of cells 402.

[0059] More particularly, each cell 402 uses a root of a generator polynomial of α^n, where n is the sequence number of the cell. For example, if there are ρ cells 402, then cells 402 can be assigned sequence numbers from zero to (ρ-1) (i.e., cell 402(0), cell 402(1), cell 402(2), ..., cell 402 (ρ-1)). In this example, cell 402(0) uses α^0, cell 402(1) uses α^1, cell 402(2) uses α^2, ..., and cell 402(ρ-1) uses $\alpha^{(\rho-1)}$. The root of a generator polynomial can be a hardwired constant, or a register set by firmware.

[0060] While the root of a generator polynomial of the plurality of cells 402 need to be sequential, the sequence can start at any number. For example, each cell 402 can use a root of a generator polynomial of α^{n+l}, where n is the sequence number of the cell and l is a fixed constant for each cell 402.

[0061] Because each cell 402 uses a different root of a generator polynomial, after process 600 (Fig. 6) is terminated, when all of the data blocks and redundancy blocks in entity 202 (Fig. 2) have been processed, different partial syndromes have been generated by each cell 402 and stored in the queues 504 of cells 402. For example, if there are ρ cells 402 and cells 402 are assigned sequence numbers from zero to (ρ-1) (i.e., cell 402(0), cell 402(1), cell 402(2), ..., cell 402 (ρ-1)), cell 402(0) generates partial syndrome S_0, cell 402(1) generates partial syndrome S_1, ..., and cell 402(ρ-1) generates partial syndrome $S_{(\rho-1)}$.

[0062] After read process 600 (Fig. 6) has been terminated, each cell 402 generates a partial error value based on the generated partial syndrome and a portion of the inverse of the error location matrix. With reference to Fig. 8, an exemplary write process 800 of generating a partial error value in cell 402 (Fig. 4) is depicted.

[0063] In 802, cell 402 (Fig. 4) is set to a write mode. With reference to Fig. 5, in the present exemplary embodiment, input logic 502 is set to receive inputs from feedback line 516, which is connected to queue 504, which holds the partial syndrome generated in cell 400 during the previous read process. More particularly, with reference to Fig. 7, multiplexer 702 is set to receive input from feedback line 516. With reference to Fig. 5, multiplier 506 is set to receive inputs from queue 504 and queue 512, which holds a portion of the inverse of the error location matrix. More particularly, multiplexer 508 with inputs connected to queues 510 and 512 is set to receive input from queue 512.

[0064] As described above, an error location matrix identifies the locations of any data blocks retrieved from the storage medium having errors. For example, an exemplary error location matrix can be expressed as:

$$\begin{bmatrix} 1 & X_0 & \cdots & X_0^{\rho-1} \\ 1 & X_1 & & X_1^{\rho-1} \\ \vdots & & \ddots & \vdots \\ 1 & X_{\rho-1} & \cdots & X_{\rho-1}^{\rho-1} \end{bmatrix},$$

where X is the location of an error and ρ is the total number of data blocks and redundancy blocks in error. The inverse of the error location matrix can be expressed as follows:

$$\begin{bmatrix} U_{0,0} & U_{1,0} & \cdots & U_{\rho-1,0} \\ U_{0,1} & U_{1,1} & & U_{\rho-1,1} \\ \vdots & & \ddots & \vdots \\ U_{0,\rho-1} & U_{1,\rho-1} & \cdots & U_{\rho-1,\rho-1} \end{bmatrix}.$$

As also described above, in the present exemplary embodiment, the inverse of the error location matrix can be generated by firmware or hardware.

[0065] In the present exemplary embodiment, each cell 402 uses a different portion of the inverse of the error location matrix. For example, with reference to Fig. 4, if there are ρ cells 402 assigned sequence numbers from zero to (ρ-1) (i.e., cell 402(0), cell 402(1), cell 402(2), ..., cell 402 (ρ-1)), cell 402(0) uses the first row of the inverse of the error location matrix (i.e., $U_{0,0}$, $U_{1,0}$, ..., $U_{\rho-1,0}$), cell 402(1) uses the second row of the inverse of the error location matrix (i.e., $U_{0,1}$, $U_{1,1}$, ..., $U_{\rho-1,1}$), ..., and cell 402(ρ-1) uses the (ρ-1) row of the inverse of the error location matrix (i.e., $U_{0,\rho-1}$, $U_{1,\rho-1}$, ..., $U_{\rho-1,\rho-1}$).

[0066] With reference to Fig. 5, in 804 (Fig. 8), multiplier 506 generates a partial error value by multiplying the partial syndrome, which is held in queue 504, with an element of the portion of the inverse of the error location matrix, which is held in queue 512. For example, with reference to Fig. 4, assume again that there are ρ cells 402 assigned sequence numbers from zero to (ρ-1) (i.e., cell 402(0), cell 402(1), cell 402(2), ..., cell 402 (ρ-1)). In cell 402(0), a first partial error value $y_0(0)$ can be generated by multiplying partial syndrome S_0 with the first element from the first row of the inverse of the error location matrix $U_{0,0}$. In cell 402(1), a second partial error value $y_0(1)$ can be generated by multiplying partial syndrome S_1 with the first element from the second row of the inverse of the error location matrix $U_{0,1}$. In cell 402(ρ-1), a (ρ-1) partial error value $y_0(\rho-1)$ can be generated by multiplying partial syndrome $S_{(\rho-1)}$ with the first element from the *(ρ-1)* row of the inverse of the error location matrix $U_{0,\rho-1}$.

[0067] With reference to Fig. 8, in 806, the partial error value generated by cell 402 (Fig. 4) is sent to summation logic 404 (Fig. 4). With reference to Fig. 4, in the example described above, the partial error values $y_0(0)$, $y_0(1)$, ..., and $y_0(\rho-1)$ generated by cells 402(0), 402(1), ..., and 402(ρ-1) are sent to summation logic 404.

[0068] To generate an error value, summation logic 404 sums the partial error values generated by cells 402. In the example described above, a first error value Y_0 is generated by summing the partial error values $y_0(0)$, $y_0(1)$, ..., and $y_0(\rho-1)$ generated by cells 402(0), 402(1), ..., and 402(ρ-1).

[0069] With reference to Fig. 9, in the present exemplary embodiment, summation logic 404 can include an array of XOR gates 902 that computes the Galois Field sum of the outputs of cells 402 (Fig. 4). It should be recognized, however, that summation logic 404 can include various components, including various types and numbers of logic components, to compute the Galois Field sum of the outputs of cells 402 (Fig. 4).

[0070] The generated error value can then be written to data buffer 114 (Fig. 1) to correct the data in the location corresponding to the elements of a portion of the inverse of the error location matrix used to generate the partial error values in cells 402. With reference to Fig. 4, in the example described above, partial error values $y_0(0)$, $y_0(1)$, ..., and $y_0(\rho-1)$ are generated in cells 402(0), 402(1), ..., and 402(ρ-1) using elements $U_{0,0}$, $U_{0,1}$, ..., and $U_{0,\rho-1}$ of a portion of the inverse of the error location matrix. With reference to Fig. 3, assume for the sake of example that these elements correspond to a portion of data block D_4. Error value Y_0 can then be written to data buffer 114 (Fig. 1) to correct the portion of data block D_4.

[0071] With reference to Fig. 4, when the size of the cache burst is greater than the amount of data processed at a time by cell 402, during write process 800 (Fig. 8), cell 402 cycles through the cache burst to process the entire cache burst. For example, as described above, cells 402 can receive data from data buffer 114 (Fig. 1) in a cache burst of 32 bytes and process one long word of the cache burst at a time. As also described above, queue 504 (Fig. 5) can include eight entries that are each 32 bits (one long word)

in length. Thus, in the example above of using element $U_{0,0}$ to generate partial error value $y_0(0)$ in cell 402(0), multiplier 506 in cell 402(0) multiplies each entry in queue 504 (Fig. 5) with element $U_{0,0}$ to produce eight sets of partial error value $y_0(0)$. Similarly, eight sets of partial error values $y_0(1)$, ..., and $y_0(\rho\text{-}1)$ are generated in cells 402(1), ..., and 402(ρ-1) using elements $U_{0,1}$, ..., and $U_{0,\rho-1}$.

[0072] Summation logic 404 also generates eight sets of error values Y_0, which are then written as a cache burst back to data buffer 114 (Fig. 1). For example, with reference to Fig. 3, assume elements $U_{0,0}$, $U_{0,1}$, ..., and $U_{0,\rho-1}$ correspond to the first 32 bytes of data block D_4, which corresponds to the first 32 code words of data block D_4, the eights sets of error values Y_0 are written to the correct the first 32 bytes of data block D_4.

[0073] With reference to Fig. 5, write process 800 (Fig. 8) can then be iterated to generate another set of partial error values, which can be summed to generate another error value, which can then be used to correct a portion of another data block with errors. For example, in a second iteration, a second set of partial error values are generated and summed to generate a second error value.

[0074] More particularly, with reference to Fig. 4, in the second iteration of 804 (Fig. 8), in cell 402(0), a first partial error value $y_1(0)$ can be generated by multiplying partial syndrome S_0 with the second element from the first row of the inverse of the error location matrix $U_{1,0}$. In cell 402(1), a second partial error value $y_1(1)$ can be generated by multiplying partial syndrome S_1 with the second element from the second row of the inverse of the error location matrix $U_{1,1}$. In cell 402(ρ-1), a (ρ-1) partial error value $y_0(\rho\text{-}1)$ can be generated by multiplying partial syndrome $S_{(\rho-1)}$ with the second element from the (ρ-1) row of the inverse of the error location matrix $U_{1,\rho-1}$.

[0075] In the second iteration of 806 (Fig. 8), the second set of partial error values $y_1(0)$, $y_1(1)$, ..., and $y_1(\rho\text{-}1)$ generated by cells 402(0), 402(1), ..., and 402(ρ-1) are sent to summation logic 404. A second error value Y_1 is then generated by summing the partial error values $y_1(0)$, $y_1(1)$, ..., and $y_1(\rho\text{-}1)$.

[0076] By iterating write process 800 (Fig. 8) and summing the generated sets of partial error values, error values can be generated to correct portions the data blocks identified as having error. The number of times write process 800 (Fig. 8) is iterated and the number of error values generated can be determined based on the number of data blocks identified as having errors (i.e., ρ). Alternatively, the number of times write process 800 (Fig. 8) is iterated and the number of error values generated can be set to the maximum number of data blocks that can be corrected, which corresponds to the number of redundancy blocks in entity 202 (Fig. 2).

[0077] After write process 800 (Fig. 8) has been completed, read process 600 (Fig. 6) can be iterated again to process another portion of the data blocks and redundancy blocks stored in data buffer 114 (Fig. 1), and then write process 800 (Fig. 8) can be iterated again to correct errors in another portion of the data blocks having errors. In the present exemplary embodiment, before iterating read process 600 (Fig. 6) again, queue 504 (Fig. 5) is cleared.

[0078] For example, with reference to Fig. 3, assume that portions of entity 202 are read in cache burst of 32 bytes in length. Read process 600 (Fig. 6) is iterated to read and process the first 32 bytes of each data block 204 and each redundancy block 206 to generate partial syndromes. Write process 800 (Fig. 8) is then iterated to generate error values and correct errors in the first 32 bytes of any data blocks 204 having errors. Queue 504 (Fig. 5) is cleared, then read process 600 (Fig. 6) is iterated again to read and process the second 32 bytes of each data block 204 and each redundancy block 206 to generate another set of partial syndromes. Write process 800 (Fig. 8) is then iterated again to generate error values and correct errors in the second 32 byte of any data blocks 204 having errors. In this manner, all the portions of each data block 204 and redundancy block 206 are read and processed using read process 600 (Fig. 6), and all the portions of any data block having errors is corrected using write process 800 (Fig. 8).

[0079] Although exemplary embodiments have been described, various modifications can be made without departing from the spirit and/or scope of the present invention. For example, with reference to Fig. 4, although decoder 4 has been described

in connection with a storage device, it should be recognized that decoder 4 can be used in connection with various devices and in various applications where data is received through a channel that introduces error, such as part of a telecommunication system. Therefore, the present invention should not be construed as being limited to the specific forms shown in the drawings and described above.